1. Query 1 takes 0.6 seconds. Query 2 takes 1.75 seconds. Query 3 takes 2.75 seconds.
2. In this lab, I implemented three operators (Filter, Join, and Aggregate) and helping classes in part 1, supported insertion and deletion of tuples and two operators (Insert, Delete) in part 2.
    - Filter: Before implementing Filter class, I implemented Predicate whose filter(Tuple t) function checks if the tuple satisfies the predicate. For a Filter operator, it takes in a Predicate and a child operator as fields. It iteratively reads in a tuple from the child operator, and applies the tuple with the Predicate until a tuple matches the Predicate or there is no tuple in the child operator. If it finds one, then return the tuple. Otherwise, return null. Filter operator does not have any I/O costs since it is on-the-fly, as expected.
    - Join: The helper class Join Predicate takes in two fields numbers from two relations and a predicate operator, checking whether two field values match on the predicate. A Join operator takes in a Join Predicate and two child operators.I implemented a hash join for equality join and tuple-at-a-time nested-loop join for other kinds of join. The hash join implementation will be discussed in the next section. For the nested-loop join, I use a Tuple field to store the current tuple in the outer loop. The outer tuple is null only at the first call of fetchNext(), so the first if statement handles the case. Then, every time fetching the next joined tuple, the join operator iterates tuples in the inner loop. If it finds one satisfies the Join Predicate, two tuples are joined by the outer tuple's fields followed by the inner tuple's fields and returned. If not, fetching the next outer tuple, and iterating the inner tuples as above. If there is no tuple in the outer loop, return null to indicate there is no joined tuple any more.
    - Aggregate: The lab separates the Aggregate operator into IntAggregator and StringAggregator based on the type of the field being aggregated over. IntAggregator supports {MIN, MAX, SUM, AVG, COUNT} operations, but StringAggregator only supports {COUNT} operation. For each kind of Aggregators, I created a map mapping the group-by field to the result of the aggregated field after a certain operation. There's an exception when dealing with AVG in IntAggregator that I kept track of the sum as values in the previous map, and created a new map to map the group-by field to the count. Then, every time the Aggregate opIterator fetches next tuple, it gets an entry from the iterator of the map.entrySet(), creates a 2-column (or 1-column if no grouping) tuple (key, value), returns the tuple.
    - Insert/Delete in HeapPage:
        - In the insertTuple, I checked if the page is full and if the tuple being inserted has the proper tuple descriptor, per the method spec. Then, I looped over the bitmap to find an empty slot by calling isSlotUsed. It is guaranteed to find one available slot because the page is not full. I added the tuple to the empty slot of the tuples array, and marked this slot used in bitmap by the markSlotUsed function. The last step in insertion is to set the tuple's record id with this pageId and slot index.
        - In the deleteTuple, I first checked if the given tuple has non-null record id to prevent NullPointerException in following codes. Then I check if the tuple is in this page by looking at the pageId in its record id and if the slot

shown in the record id is already empty. Then, I set the slot in tuples to be null and mark the slot in the bitmap unused by markSlotUsed.

- Insert/Delete in HeapFile:
    - For insertTuple, I looped over the pages in that file until a page had empty slots. If there is a non-full page, call HeapPage insertTuple, and return an ArrayList containing the page. If there isn't, I need to append a new page to the file. I initialize a new HeapPage with empty data, call writePage() to append the heap page to the file on disk, call BufferPool.getPage to retrieve that page and cache it, call HeapPage insertTuple, and return an ArrayList containing the page.
    - For deleteTuple, I check for non-null record id as well, and check if the pageId in the record id belongs to this file. Then, I retrieve the page by calling BufferPool.getPage() on the pageId in its record id, and call the HeapPage deleteTuple.
- Insert/Delete in BufferPool:
    - For insertTuple, I find the heapfile with the given table id by calling Database.getCatalog().getDatabaseFile and call HeapFile insertTuple on the given tuple. Then, for each modified page, mark it dirty by calling HeapPage markDirty and add the page to cache and the LRU list.
    - For deleteTuple, I find the heapfile with the table id in the tuple's record id, and call HeapFile deleteTuple on the given tuple. Then, for each modified page, mark it dirty by calling HeapPage markDirty and add the page to cache and the LRU list.
- Evict page in BufferPool: I implement LRU eviction policy which will be described in the next section. In large, evictPage() gets the LRU pageId, flushes the page and removes it from the buffer pool.

3. In the implementation of hash join, I have a map to store tuples from child 1 hashed on the specific field, a Tuple to store the current tuple from child 2 on which the iterator is, and a Tuple Iterator to keep track of whether there are matched tuple not joined with the current tuple. In fetchNext(), I first check if there is any unjoined matched tuple. If not, I get the next tuple from child 2 and the corresponding hashing bucket. If there is a matched tuple, join them together and return.
   I used the LRU page eviction policy. To implement it, I have a LinkedList<PageId> field in the buffer pool to store the information about which pageId is least recently used. I let the first node be LRU and the last node be most recently used. So when getPage(pid) is called, I removed the pageId from the LRU list first and added it back to the end of the list. In insertTuple and deleteTuple in the BufferPool, for each modified page, I also removed its pid from the LRU list first and appended it to the list again. In this way, the first pageId in the LRU list is the least recently used one among pageIds in the bufferpool.
4. Add a test for testing that Insert and Delete operators can only return one tuple because they insert or delete all tuples from the child operator in a single fetchNext().
5. No changes made to the API.
6. All completed.